IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
BOARD OF PATENT APPEALS AND INTERFERENCES

In re the Application of: Martin D. Richek	:	Group Art Unit: 2124
Serial No.: 10/079,928	:	Examiner: Satish Rampuria
Filed: February 19, 2002	:	
For: TYPE-SAFE HOMOGENEOUS LINKAGE FOR HETEROGENEOUS SMART POINTERS	:	

APPLICANT'S APPEAL BRIEF

INTRODUCTION

This appeal is from the Primary Examiner's final rejection, dated April 21, 2006, in the above-identified patent application.

1. REAL PARTY IN INTEREST

The real party in interest is Assignee, Quazant Technology, Inc. of Cypress, TX.

2. RELATED APPEALS AND INTERFERENCES

Applicant does not know of any prior or pending appeals, interferences, or judicial proceedings which will directly affect or be directly affected by or have a bearing on the Board's decision in the present appeal.

3. STATUS OF CLAIMS

Claims 3-10, 12, 13, 15-23, 25-28, and 30-35 are pending in this application. All of the pending claims stand finally rejected; no claim is allowed.

4. STATUS OF AMENDMENTS

Applicant is waiting for an advisory action confirming entry of the amendments presented in the paper mailed August 8, 2006. However, Examiner Rampuria confirmed by phone on October 24, 2006 that the August 8, 2006 amendment would be entered. Thus, the attached claims reflect the claims as amended in the August 8, 2006 amendment.

5. SUMMARY OF CLAIMED SUBJECT MATTER

For each of the independent claims on appeal, Applicant's invention relates to problems encountered in object-oriented programming involving the control of the lifetime of dynamically allocated objects. These problems occur in multiple forms, one of which is a dangling pointer, where reference is made to an object after its memory has been released to the system (i.e., the object is no longer available). Where a reference is made to a dangling pointer, the run-time result is unpredictable and may result in an invalid value being returned, crashing of the program, or both. (Application at [0003]-[0004].) (As argued below, this problem is present in the art cited against the appealed claims, and highlights one of the differences between the cited art and Applicant's claimed invention which overcomes this problem.)

In addition, in many instances memory allocation relates to a number of pointers some of which may point not only to a particular object, but also to differently-typed sub-objects of the object. The pointers to sub-objects, like pointers to the object containing the sub-objects, also need to be monitored to avoid the dangling pointer condition. Such monitoring needs to be maintained as pointers to differently-typed sub-objects are assigned to one another. Thus, in additional aspects, Applicant's claimed invention addresses the need for programming tools which also enable management of heterogeneous pointers. (Application at [0007].)

Claim 3

Applicant's invention as recited in independent claim 3 relates to a computer-implemented method of memory management. The method includes the steps of "providing a smart pointer for association with a memory-resident element, the smart pointer including a next pointer" and "providing an assignment means for assigning the next pointer to point to the smart pointer thereby creating a linked list comprising the smart pointer." As used in the Application, "a smart pointer means a user-defined type that includes the behavior of a built-in pointer and adds additional functions." (Application at [0008]). The smart pointer includes a next pointer for pointing to a selected smart pointer, which may be a different smart pointer on the linked list or may be the selected smart pointer itself.

The method also includes the step of "providing a comparison means for comparing the value of the next pointer to the value of the memory location of the smart pointer in which the next pointer is included, whereby a determination can be made if the linked list contains more than one smart pointer." If the value of the memory location of the "the smart pointer in which the next pointer is included" equals the value of the "value of the next pointer of the smart pointer," then the linked list necessarily comprises only the smart pointer, and the memory-resident element can be deleted upon deletion of the smart pointer. In this regard, the method provides the step of "deleting the memory-resident element associated with the smart pointer if the value of the next pointer of the smart pointer is equal to the value of the memory location of the smart pointer in which the next pointer is included and not deleting the memory-resident element if the value of the next pointer of the smart pointer is not equal to the value of the memory location of the smart pointer in which the next pointer is included." This test for deletion of the memory-resident element ensures that the memory-resident element is only deleted upon deletion of the last smart pointer to point to the memory-resident element. As such, the situation of the dangling pointer, as discussed above, is avoided. This test is not present in the art used to reject the claims (as explained below in the arguments section).

The operation of this test in Applicant's invention is exemplified for a ring list, at paragraphs [0020]-[0022] of the Application, where the "object" is an example of a "memory-resident element" as recited in claim 3:

"When a smart pointer gives up its object pointer — whether it is receiving a new object pointer or it is expiring— then, that smart pointer is removed from the ring of which it is a member. In particular, to know when an object is no longer needed and the object's memory should therefore be released, it is important to test whether the smart pointer giving up its object pointer is the last member of the ring. If the smart pointer is the last member of the ring, the object to which it points may be deleted. The 'last member of the ring' test is illustrated in Fig. 4.

A smart pointer that is alone on a ring points to itself as both the next and the previous 'members' of the ring, as illustrated in Fig. 3. That is, the 'Next' pointer 16 and 'Previous' pointer 14 of the first smart pointer 10 each point to the first smart pointer 10. Thus, the values of the 'Next' pointer 16 and 'Previous' pointer 14 are equal, and each contains the memory address of the smart pointer 10 to which it belongs. More importantly, a significant distinction exists between the 'Next' and 'Previous' pointer values of each smart pointer in the single-member ring 100 of Fig. 3 and the two-member ring 100 of Fig. 1. In the two-member ring 100, the 'Next' pointer 16 and the 'Previous' pointer of the each smart pointer contain the memory address of the other smart pointer. In contrast, for the single-member ring 100, the 'Next' pointer 16 and the 'Previous' pointer 14 of the first smart pointer 10 each contains the memory address of the smart pointer to which it belongs. Thus, while the values of the 'Next' pointer 16 and the 'Previous' pointer 14 are equal to each other in both the single-member ring and the two-member ring, the values of the 'Next' pointer 16 and the 'Previous' pointer 14 are equal to the address of the smart pointer to which they belong only when the ring 100 contains a single-member as shown in Fig. 3. Therefore, the test for a single-member ring, step 410 of Fig. 4, comprises comparing the value of either the 'Next' pointer 16 or the 'Previous' pointer 14 to the address of the smart pointer to which the 'Next' pointer 16 and 'Previous' pointer 14 belong, e.g., the first smart pointer 10. If the test for equality between the value of the 'Next' pointer 16 and the address of the first smart pointer 10, as provided at step 410, is true, then the object 30 is deleted at step 420. In the C++ embodiment of the invention of the Example below, the test of step 410 is provided by the member function *bool IsOnly () const*.

If the test of step 410 returns 'false', then the ring contains more than one smart pointer, and the smart pointer which is giving up its object pointer is removed from the ring. For example, if the ring contains three elements as shown in Fig. 2 and the smart pointer which is giving up its object pointer is the third smart pointer 40, then the third smart pointer 40 is removed from the ring 100 yielding the two-member ring shown in Fig. 1. The 'Next' pointer 16 of the first pointer 10 and 'Previous' pointer 24 of the second smart pointer 20 are set to point to the second smart pointer 20 and the first smart pointer 10, respectively, as explained above with reference to Fig. 1. In the C++ embodiment of the invention of the Example below, detaching a smart pointer from a ring is provided by the member function *void Detach () const*." (Application at [0020]-[0022]. Emphasis Added.)

The invention of claim 3 includes112 paragraph 6 means-plus-function elements, *i.e.*, "assignment means for assigning..." and "comparison means for comparing...". The algorithm structures and acts related to the "means for assigning" are disclosed in the specification at paragraph [0019] along with the figures and examples cited therein. The algorithm structures and acts related to the "means for comparing" are disclosed in the specification at paragraph [0021]-[0022] along with the figures (*e.g.*, step 410 of Fig. 4) and examples cited therein.

Claim 13

Applicant's invention as recited in independent claim 13 relates to a computer-implemented method of memory management. The method includes the steps of "providing a linked list comprising a smart pointer associated with a memory-resident element, the smart pointer including a next-pointer for pointing to the smart pointer" and "providing a comparison means for comparing the value of the memory location of the smart pointer to the value of the next-pointer of the smart pointer, to provide a determination whether the linked list contains only the smart pointer." If the value of "the memory location of the smart pointer" equals the"value of the next-pointer of the smart pointer," then the linked list necessarily comprises only the smart pointer, and the memory-resident element can be deleted upon deletion of the smart pointer. In this regard, the method provides the step of "deleting the memory-resident element associated with the smart pointer if the value of the next-pointer of the smart pointer is equal to the value of the memory location of the smart pointer in which the next-pointer is included and not deleting the memory-resident element if the value of the next-pointer of the smart pointer is not equal to the value of the memory location of the smart pointer in which the next-pointer is included." This test for deletion of the memory-resident element ensures that the memory-resident element is only deleted upon deletion of the last smart pointer to point to the memory-resident element. As such, the situation of the dangling pointer, as discussed above, is avoided. This test is not present in the art used to reject the claims (as explained below in the arguments section).

The operation of this test in Applicant's invention is exemplified for a ring list, at paragraphs [0020]-[0022] of the Application, where the "object" is an example of a "memory-resident element" as recited in claim 13:

"When a smart pointer gives up its object pointer — whether it is receiving a new object pointer or it is expiring— then, that smart pointer is removed from the ring of which it is a member. In particular, to know when an object is no longer needed and the object's memory should therefore be released, it is important to test whether the smart pointer giving up its object pointer is the last member of the ring. If the smart pointer is the last member of the ring, the object to which it points may be deleted. The 'last member of the ring' test is illustrated in Fig. 4.

A smart pointer that is alone on a ring points to itself as both the next and the previous 'members' of the ring, as illustrated in Fig. 3. That is, the 'Next' pointer 16 and 'Previous' pointer 14 of the first smart pointer 10 each point to the first smart pointer 10. Thus, the values of the 'Next' pointer 16 and 'Previous' pointer 14 are equal, and each contains the memory address of the smart pointer 10 to which it belongs. More importantly, a significant distinction exists between the 'Next' and 'Previous' pointer values of each smart pointer in the single-member ring 100 of Fig. 3 and the two-member ring 100 of Fig. 1. In the two-member ring 100, the 'Next' pointer 16 and the 'Previous' pointer of the each smart pointer contain the memory address of the other smart pointer. In contrast, for the single-member ring 100, the 'Next' pointer 16 and the 'Previous' pointer 14 of the first smart pointer 10 each contains the memory address of the smart pointer to which it belongs. Thus, while the values of the 'Next' pointer 16 and the 'Previous' pointer 14 are equal to each other in both the single-member ring and the two-member ring, the values of the 'Next' pointer 16 and the 'Previous' pointer 14 are equal to the address of the smart pointer to which they belong only when the ring 100 contains a single-member as shown in Fig. 3. Therefore, the test for a single-member ring, step 410 of Fig. 4, comprises comparing the value of either the 'Next' pointer 16 or the 'Previous' pointer 14 to the address of the smart pointer to which the 'Next' pointer 16 and 'Previous' pointer 14 belong, e.g., the first smart pointer 10. If the test for equality between the value of the 'Next' pointer 16 and the address of the first smart pointer 10, as provided at step 410, is true, then the object 30 is deleted at step 420. In the C++ embodiment of the invention of the Example below, the test of step 410 is provided by the member function *bool IsOnly () const*.

If the test of step 410 returns 'false', then the ring contains more than one smart pointer, and the smart pointer which is giving up its object pointer is removed from the ring. For example, if the ring contains three elements as shown in Fig. 2 and the smart pointer which is giving up its object pointer is the third smart pointer 40, then the third smart pointer 40 is removed from the ring 100 yielding the two-member ring shown in Fig. 1. The 'Next' pointer 16 of the first pointer 10 and 'Previous' pointer 24 of the second smart pointer 20 are set to point to the second smart pointer 20 and the first smart pointer 10, respectively, as explained above with reference to Fig. 1. In the C++ embodiment of the invention of the Example below, detaching a smart pointer from a ring is provided by the member function *void Detach () const*." (Application at [0020]-[0022]. Emphasis Added.)

The invention of claim 3 includes112 paragraph 6 means-plus-function elements, *i.e.*,"comparison means for comparing...". The algorithm structures and acts related to the "means for comparing" are disclosed in the specification at paragraph [0021]-[0022] along with the figures (*e.g.*, step 410 of Fig. 4) and examples cited therein.

Claim 23

Applicant's invention as recited in independent claim 23 relates to a computer-implemented method of memory management. The method includes the step of "providing a linked list comprising a first smart pointer and a second smart pointer each associated with a memory-resident element, the first smart pointer including a first next-pointer for pointing to the second smart pointer and the second smart pointer including a second next-pointer for pointing to the first smart pointer."

The claimed method also includes the step of "providing a comparison means for comparing the value of the memory location of a selected smart pointer giving up its association with the memory-resident element to the value of the next-pointer of the selected smart pointer, to provide a determination whether the linked list contains only the selected smart pointer." If the "value of the memory location of the selected smart pointer" equals the "value of the next pointer of the selected smart pointer," then the linked list necessarily comprises only the smart pointer, and the memory-resident element can be deleted upon deletion of the smart pointer. In this regard, the method provides the step of "providing a deletion means for deleting the memory-resident element associated with the smart pointer if the value of the next-pointer of the selected smart pointer is equal to the value of the memory location of the selected smart pointer in which the next-pointer is included and not deleting the memory-resident element if the value of the next- pointer of the selected smart pointer is not equal to the value of the memory location of the selected smart pointer in which the next pointer is included." This test for deletion of the memory-resident element ensures that the memory-resident element is only deleted upon deletion of the last smart pointer to point to the memory-resident element. As such, the situation of the dangling pointer, as discussed above, is avoided. This test is not present in the art used to reject the claims (as explained below in the arguments section).

The operation of this test in Applicant's invention is exemplified for a ring list, at paragraphs [0020]-[0022] of the Application, where the "object" is an example of a "memory-resident element" as recited in claim 23:

"When a smart pointer gives up its object pointer — whether it is receiving a new object pointer or it is expiring— then, that smart pointer is removed from the ring of which it is a member. In particular, to know when an object is no longer needed and the object's memory should therefore be released, it is important to test whether the smart pointer giving up its object pointer is the last member of the ring. If the smart pointer is the last member of the ring, the object to which it points may be deleted. The 'last member of the ring' test is illustrated in Fig. 4.

A smart pointer that is alone on a ring points to itself as both the next and the previous 'members' of the ring, as illustrated in Fig. 3. That is, the 'Next' pointer 16 and 'Previous' pointer 14 of the first smart pointer 10 each point to the first smart pointer 10. Thus, the values of the 'Next' pointer 16 and 'Previous' pointer 14 are equal, and each contains the memory address of the smart pointer 10 to which it belongs. More importantly, a significant distinction exists between the 'Next' and 'Previous' pointer values of each smart pointer in the single-member ring 100 of Fig. 3 and the two-member ring 100 of Fig. 1. In the two-member ring 100, the 'Next' pointer 16 and the 'Previous' pointer of the each smart pointer contain the memory address of the other smart pointer. In contrast, for the single-member ring 100, the 'Next' pointer 16 and the 'Previous' pointer 14 of the first smart pointer 10 each contains the memory address of the smart pointer to which it belongs. <u>Thus, while the values of the 'Next' pointer 16 and the 'Previous' pointer 14 are equal to each other in both the single-member ring and the two-member ring, the values of the 'Next' pointer 16 and the 'Previous' pointer 14 are equal to the address of the smart pointer to which they belong only when the ring 100 contains a single-member as shown in Fig. 3. Therefore, the test for a single-member ring, step 410 of Fig. 4, comprises comparing the value of either the 'Next' pointer 16 or the 'Previous' pointer 14 to the address of the smart pointer to which the 'Next' pointer 16 and 'Previous' pointer 14 belong, e.g., the first smart pointer 10.</u> If the test for equality between the value of the 'Next' pointer 16 and the address of the first smart pointer 10, as provided at step 410, is true, then the object 30 is deleted at step 420. In the C++ embodiment of the invention of the Example below, the test of step 410 is provided by the member function *bool IsOnly () const.*

If the test of step 410 returns 'false', then the ring contains more than one smart pointer, and the smart pointer which is giving up its object pointer is removed from the ring. For example, if the ring contains three elements as shown in Fig. 2 and the smart pointer which is giving up its object pointer is the third smart pointer 40, then the third smart pointer 40 is removed from the ring 100 yielding the two-member ring shown in Fig. 1. The 'Next' pointer 16 of the first pointer 10 and 'Previous' pointer 24 of the second smart pointer 20 are set to point to the second smart pointer 20 and the first smart pointer 10, respectively, as explained above with reference to Fig. 1. In the C++ embodiment of the invention of the Example below, detaching a smart pointer from a ring is provided by the member function *void Detach () const*." (Application at [0020]-[0022]. Emphasis Added.)

The invention of claim 23 includes112 paragraph 6 means-plus-function elements, *i.e.*, "comparison means for comparing..." and "deletion means for deleting...". The algorithm structures and acts related to the "means for comparing" are disclosed in the specification at paragraph [0021]-[0022] along with the figures (*e.g.*, step 410 of Fig. 4) and examples cited therein. The algorithm structures and acts related to the "means for deleting" are disclosed in the specification at paragraph [0030] along with the examples cited therein.

Claim 34

Applicant's invention as recited in dependent claim 34 relates to a further aspect of the invention comprising a framework to provide type-safe homogeneous linkage for heterogeneous smart pointers. The framework comprises a base common to all smart pointers and a template for managing inter-class assignment and inter-class conversion of smart pointers. All ring operations are contained in the common base shared by all smart pointers. Providing a base common to all smart pointers is desirable in class-hierarchical programming environments, such as C^{++}, where the derived pointer is typed, since provision of a common pointer base allows pointers to objects of different classes in a common class hierarchy that point to different sub-objects of the same object to be members of a single ring, providing a complete representation of all references to an object. In this regard, claim 34 includes the step of "providing a conversion means for providing automatic conversion between the first smart pointer and the second smart pointer", where"the first smart pointer is associated with a first object of a first class and the second smart pointer is associated with a second object of a second class." The invention of claim 34 includes112 paragraph 6 means-plus-function elements, *i.e.*, "conversion means." The algorithm structures and acts related to the "conversion means" are disclosed in the Application as part of the example discussed at paragraphs [0024]-[0035].

Claim 35

Applicant's invention as recited in independent claim 35 relates to a computer-implemented method of memory management. The method includes the steps of "providing a smart pointer for association with a memory-resident element, the smart pointer including a previous pointer" and "providing an assignment means for assigning the previous pointer to point to the smart pointer thereby creating a linked list comprising the smart pointer."

The method also includes the step of "providing a comparison means for comparing the value of the previous pointer to the value of the memory location of the smart pointer in which the selected previous pointer is included, whereby a determination can be made if the ring contains more than one smart pointer." If the value of the memory location of the "smart pointer in which the selected previous pointer is included" equals the "value of the previous pointer," then the linked list necessarily comprises only the smart pointer, and the memory-resident element can be deleted upon deletion of the smart pointer. In this regard, the method provides the step of "deleting the memory-resident element associated with the smart pointer if the value of the previous pointer of the smart pointer is equal to the value of the memory location of the smart pointer in which the previous pointer is included and not deleting the memory-resident element if the value of the previous pointer of the smart pointer is not equal to the value of the memory location of the smart pointer in which the previous pointer is included." This test for deletion of the memory-resident element ensures that the memory-resident element is only deleted upon deletion of the last pointer to smart point to the memory-resident element. As such, the situation of the dangling pointer, as discussed above, is avoided. This test is not present in the art used to reject the claims (as explained below in the arguments section).

The operation of this test in Applicant's invention is exemplified for a ring list, at paragraphs [0020]-[0022] of the Application, where the "object" is an example of a "memory-resident element" as recited in claim 35:

"When a smart pointer gives up its object pointer — whether it is receiving a new object pointer or it is expiring— then, that smart pointer is removed from the ring of which it is a member. In particular, to know when an object is no longer needed and the object's memory should therefore be released, it is important to test whether the smart pointer giving up its object pointer is the last member of the ring. If the smart pointer is the last member of the ring, the object to which it points may be deleted. The 'last member of the ring' test is illustrated in Fig. 4.

A smart pointer that is alone on a ring points to itself as both the next and the previous 'members' of the ring, as illustrated in Fig. 3. That is, the 'Next' pointer 16 and 'Previous' pointer 14 of the first smart pointer 10 each point to the first smart pointer 10. Thus, the values of the 'Next' pointer 16 and 'Previous' pointer 14 are equal, and each contains the memory address of the smart pointer 10 to which it belongs. More importantly, a significant distinction exists between the 'Next' and 'Previous' pointer values of each smart pointer in the single-member ring 100 of Fig. 3 and the two-member ring 100 of Fig. 1. In the two-member ring 100, the 'Next' pointer 16 and the 'Previous' pointer of the each smart pointer contain the memory address of the other smart pointer. In contrast, for the single-member ring 100, the 'Next' pointer 16 and the 'Previous' pointer 14 of the first smart pointer 10 each contains the memory address of the smart pointer to which it belongs. Thus, while the values of the 'Next' pointer 16 and the 'Previous' pointer 14 are equal to each other in both the single-member ring and the two-member ring, the values of the 'Next' pointer 16 and the 'Previous' pointer 14 are equal to the address of the smart pointer to which they belong only when the ring 100 contains a single-member as shown in Fig. 3. Therefore, the test for a single-member ring, step 410 of Fig. 4, comprises comparing the value of either the 'Next' pointer 16 or the 'Previous' pointer 14 to the address of the smart pointer to which the 'Next' pointer 16 and 'Previous' pointer 14 belong, e.g., the first smart pointer 10. If the test for equality between the value of the 'Next' pointer 16 and the address of the first smart pointer 10, as provided at step 410, is true, then the object 30 is deleted at step 420. In the C++ embodiment of the invention of the Example below, the test of step 410 is provided by the member function *bool IsOnly () const*.

If the test of step 410 returns 'false', then the ring contains more than one smart pointer, and the smart pointer which is giving up its object pointer is removed from the ring. For example, if the ring contains three elements as shown in Fig. 2 and the smart pointer which is giving up its object pointer is the third smart pointer 40, then the third smart pointer 40 is removed from the ring 100 yielding the two-member ring shown in Fig. 1. The 'Next' pointer 16 of the first pointer 10 and 'Previous' pointer 24 of the second smart pointer 20 are set to point to the second smart pointer 20 and the first smart pointer 10, respectively, as explained above with reference to Fig. 1. In the C++ embodiment of the invention of the Example below, detaching a smart pointer from a ring is provided by the member function *void Detach () const*." (Application at [0020]-[0022]. Emphasis Added.)

The invention of claim 35 includes112 paragraph 6 means-plus-function elements, *i.e.*, "assignment means for assigning..." and "comparison means for comparing...". The algorithm structures and acts related to the "means for assigning" are disclosed in the specification at paragraph [0019] along with the figures and examples cited therein. The algorithm structures and acts related to the "means for comparing" are disclosed in the specification at paragraph [0021]-[0022] along with the figures (*e.g.*, step 410 of Fig. 4) and examples cited therein.

6. GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

The rejection of claims 3-10, 12, 13, 15-23, 25-28, and 30-35 under 35 U.S.C. 102(e) as being anticipated by Oliver US 6,144,965 is to be reviewed on appeal.

7. ARGUMENT

7.1 <u>Rejections Under 35 U.S.C. 102(e)</u>

Rejections under 35 USC 102 are proper only when the claimed subject matter is identically disclosed or described in the prior art. <u>In re Arkley</u>, 172 USPQ 524 (CCPA 1972). Applying this rule of law to the present case, the rejection of claims 3-10, 12, 13, 15-23, 25-28, and 30-35 is improper, because the subject matter of these claims is not identically disclosed or described, explicitly or inherently, in the Oliver patent.

7.1.1. <u>Respecting Claims 3-10, 12</u>

For purpose of the appeal, Applicant is satisfied to let claims 4-10 and 12 stand or fall together with claim 3 from which they depend. Applicant submits that claims 3-10 and 12 are separately patentable from the other pending claims for the reasons set forth below.

Claims 3-10 and 12 stand rejected under 25 USC 102 in view of Oliver. Applicant respectfully maintains that claim 3 is patentable over Oliver for at least the reason that Oliver fails to disclose, explicitly or inherently, the feature of "providing a comparison means for comparing the value of the <u>next pointer</u> <u>to</u> the value of the <u>memory location of the smart pointer in which the next pointer is included</u>, whereby a determination can be made if the linked list contains more than one smart pointer." (Emphasis Added.) That is, Applicant's claimed test compares two memory values, that of the "next pointer" to that of the "memory location of the smart pointer in which the next pointer is included." Applicant's claimed test does <u>not</u> compare the value of the "next pointer" to that of the "previous pointer", as is done in Oliver. The Examiner concedes that Oliver compares the values of the next and previous pointers when he states that "Oliver discloses the pointers next and previous are examined (compared) before they are deleted from the memory (see col. 5, lines 30-46 and Fig. 5D step 502 and related text)" (Final Action, page 4.) For at least this reason, it is clear that Oliver fails to disclose each and every element recited in claim 3.

Turning more specifically to the portion of the Final Action applying Oliver to claim 3, the Examiner states that

> Oliver disclose:... providing a comparison means for comparing the value of the next pointer to the value of the memory location of the smart pointer in which the selected next pointer is included, whereby a determination can be made if the ring contains more than one smart pointer (col. 5, lines 34-37 'FIG. 5D ... Prior to deleting a pointer, the 'next pointer' and 'previous pointer' pointers are examined (compared) ... If the 'next pointer' pointer is the same as the 'previous pointer' then there is clearly only one pointer remaining in the list ... final pointer is deleted, no pointers will remain in the list. Thus, if the 'next pointer' pointer is the same as the 'previous pointer' pointer, then the object.. . deleted. .. and the last pointer to the object ... deleted ... If the 'next pointer' pointer is not the same as the 'previous pointer' ... other pointers remaining that point to the object. In this case, a pointer is removed from the list in step 508 and then the pointer is deleted in step 506'). (Final Action, pages 8-9.)

Thus, it is also clear from this text that the Examiner (perhaps unintentionally) acknowledges that Oliver compares the next and previous pointers. The "value of the memory location of the smart pointer in which the selected next pointer is included" is not even considered or mentioned in the cited text of Oliver. It is not used in the Oliver test. Only the next and previous pointers are compared in Oliver. Hence, Oliver unquestionably fails to disclose at least Applicant's claimed feature of "comparing the value of the next pointer to *<u>the value of the memory location of the smart pointer</u>* in which the selected next pointer is included." Likewise, Oliver fails disclose the step of "deleting the memory-resident element associated with the smart pointer if the value of the next pointer of the smart pointer is equal to the value of the memory location of the smart pointer in which the next pointer is included and not deleting the memory-resident element if the value of the next pointer of the smart pointer is not equal to the value of the memory location of the smart pointer in which the next pointer is included" as recited in claim 3. For at least these reasons, the rejection of claim 3 in view of Oliver is deficient, as Oliver fails to disclose each and every element recited in claim 3.

Indeed, Applicant has specifically explained to the public in the body of the application reasons not to compare the values of the next and previous pointers in a test for deciding when to delete the object to which a pointer points. To do so would run the risk of creating a dangling pointer, a problem sought to be solved by Applicant's invention. (Application at [004].) Hence, there are significant practical and functional differences between the test claimed and that of Oliver.

Applicant has taught the public that the values of the next and previous pointers are equal when there are two smart pointers on a ring. (Application at [0021].) In addition, Applicant has warned against deleting an object when there are two or more members on a ring, but deleting only when there is one member on the ring, otherwise a dangling pointer will remain on the ring. (Application at [0021] and [0004]). Accordingly, Applicant has taught the public not to use a test comparing the next and previous pointers, but rather a test using either the next or previous pointer and comparing it to the address of the smart pointer to which the next or previous pointer belongs:

> More importantly, a significant distinction exists between the "Next" and "Previous" pointer values of each smart pointer in the single-member ring 100 of Fig. 3 and the two-member ring 100 of Fig. 1. In the two-member ring 100, the "Next" pointer 16 and the "Previous" pointer of the each smart pointer contain the memory address of the other smart pointer. In contrast, for the single-member ring 100, the "Next" pointer 16 and the "Previous" pointer 14 of the first smart pointer 10 each contains the memory address of the smart pointer to which it belongs. Thus, while the values of the "Next" pointer 16 and the "Previous" pointer 14 are equal to each other in both the single-member ring and the two-member ring, the values of the "Next" pointer 16 and the "Previous" pointer 14 are equal to the address of the smart pointer to which they belong only when the ring 100 contains a single-member as shown in Fig. 3. *Therefore, the test for a single-member ring, step 410 of Fig. 4, comprises comparing the value of either the "Next" pointer 16 or the "Previous" pointer 14 to the address of the smart pointer to which the "Next" pointer 16 and "Previous" pointer 14 belong, e.g., the first smart pointer 10.* If the test for equality between the value of the "Next" pointer 16 and the address of the first smart pointer 10, as provided at step 410, is true, then the object 30 is deleted at step 420. (Application at [0021]. Emphasis Added.)

In sum, if one were to erroneously conclude that the Oliver test was equivalent to that claimed by Applicant, the "test" would return a 'true' result if there were two smart pointers on the linked list. Then, at the last step of claim 3 the memory-resident element would be prematurely deleted when two pointers remained on the linked list, thereby creating a dangling pointer, in contradiction to what Applicant explicitly teaches and what claim 3 expressly calls for.

Hence, for at least the above reasons, Applicant respectfully submits that Oliver fails to disclose each and every element recited in claim 3, either explicitly or inherently. Accordingly, Applicant respectfully requests that the Board reverse the Examiner's rejection of claim 3, as well as 4-10 and 12 which depend respectively therefrom.

7.1.2. <u>Respecting Claims 13, 15-22</u>

For purpose of the appeal, Applicant is satisfied to let claims 15-22 stand or fall together with claim 13 from which they depend. Applicant submits that claims 13 and 15-22 are separately patentable from the other pending claims for the reasons set forth below.

Claims 13 and 15-22 stand rejected under 25 USC 102 in view of Oliver. Applicant respectfully maintains that claim 13 is patentable over Oliver for at least the reason that Oliver fails to disclose, explicitly or inherently, the feature of "providing a comparison means for comparing the value of the <u>memory location of the smart pointer</u> <u>to</u> the value of <u>the next-pointer of the smart pointer</u>, to provide a determination whether the linked list contains only the smart pointer ." (Emphasis Added.) That is, Applicant's claimed test compares two memory values, that of the "memory location of the smart pointer" to that of the "the next-pointer of the smart pointer." Applicant's claimed test does <u>not</u> compare the value of the "next pointer" to that of the "previous pointer", as is done in Oliver. The Examiner concedes that Oliver compares the values of the next and previous pointers when he states that "Oliver discloses the pointers next and previous are examined (compared) before they are deleted from the memory (see col. 5, lines 30-46 and Fig. 5D step 502 and related text)" (Final Action, page 4.) For at least this reason, it is clear that Oliver fails to disclose each and every element recited in claim 13.

Turning more specifically to the portion of the Final Action applying Oliver to claim 13, the Examiner states that

> Oliver disclose:... providing a comparison means for comparing the value of memory of the smart pointer to the value of the next pointer of the smart pointer, to provide whether the linked list

> contains only the smart pointer (col. 5, lines 34-37 "pointer ... examined ... pointer is the same ... one pointer remaining in the list"). (Final Action at 12.)

But, as the Examiner already acknowledged, the cited text of Oliver at col. 5, lines 34-37 compares the next and previous pointers. The "value of the memory location of the smart pointer" is not even considered or mentioned in the cited text of Oliver. It is not used in the Oliver test. Only the next and previous pointers are compared in Oliver. Hence, Oliver unquestionably fails to disclose at least Applicant's claimed feature of "comparing the value of the <u>memory location of the smart pointer</u> <u>to</u> the value of the next-pointer of the smart pointer." Likewise, Oliver fails disclose the step of "deleting the memory-resident element associated with the smart pointer if the value of the next-pointer of the smart pointer is equal to the value of the memory location of the smart pointer in which the next-pointer is included and not deleting the memory-resident element if the value of the next-pointer of the smart pointer is not equal to the value of the memory location of the smart pointer in which the next-pointer is included" as recited in claim 13. For at least these reasons, the rejection of claim 13 in view of Oliver is deficient, as Oliver fails to disclose each and every element recited in claim 13.

Indeed, Applicant has specifically explained to the public in the body of the application reasons not to compare the values of the next and previous pointers in a test for deciding when to delete the object to which a pointer points. To do so would run the risk of creating a dangling pointer, a problem sought to be solved by Applicant's invention. (Application at [004].) Hence, there are significant practical and functional differences between the test claimed and that of Oliver.

Applicant has taught the public that the values of the next and previous pointers are equal when there are two smart pointers on a ring. (Application at [0021].) In addition, Applicant has warned against deleting an object when there are two or more members on a ring, but deleting only when there is one member on the ring, otherwise a dangling pointer will remain on the ring. (Application at [0021] and [0004]). Accordingly, Applicant has taught the public not to use a test comparing the next and previous pointers, but rather a test using either the next or previous pointer and comparing it to the address of the smart pointer to which the next or previous pointer belongs:

> More importantly, a <u>significant distinction exists between the "Next" and "Previous" pointer values of each smart pointer in the single-member ring 100 of Fig. 3 and the two-member ring 100</u> of Fig. 1. In the two-member ring 100, the "Next" pointer 16 and

> the "Previous" pointer of the each smart pointer contain the memory address of the other smart pointer. In contrast, for the single-member ring 100, the "Next" pointer 16 and the "Previous" pointer 14 of the first smart pointer 10 each contains the memory address of the smart pointer to which it belongs. Thus, while the values of the "Next" pointer 16 and the "Previous" pointer 14 are equal to each other in both the single-member ring and the two-member ring, the values of the "Next" pointer 16 and the "Previous" pointer 14 are equal to the address of the smart pointer to which they belong only when the ring 100 contains a single-member as shown in Fig. 3. *Therefore, the test for a single-member ring, step 410 of Fig. 4, comprises comparing the value of* either *the "Next" pointer 16 or the "Previous" pointer 14 to the address of the smart pointer to which the "Next" pointer 16 and "Previous" pointer 14 belong, e.g., the first smart pointer 10.* If the test for equality between the value of the "Next" pointer 16 and the address of the first smart pointer 10, as provided at step 410, is true, then the object 30 is deleted at step 420. (Application at [0021]. Emphasis Added.)

In sum, if one were to erroneously conclude that the Oliver test was equivalent to that claimed by Applicant, the "test" would return a 'true' result if there were two smart pointers on the linked list. Then, at the last step of claim 13 the memory-resident element would be prematurely deleted when two pointers remained on the linked list, thereby creating a dangling pointer, in contradiction to what Applicant explicitly teaches and what claim 13 expressly calls for.

Hence, for at least the above reasons, Applicant respectfully submits that Oliver fails to disclose each and every element recited in claim 13, either explicitly or inherently. Accordingly, Applicant respectfully requests that the Board reverse the Examiner's rejection of claim 13, as well as 15-22 which depend respectively therefrom.

7.1.3. Respecting Claims 23, 25-28, and 30-33

For purpose of the appeal, Applicant is satisfied to let claims 25-28 and 30-33 stand or fall together with claim 23 from which they depend. Applicant submits that claims 23, 25-28, and 30-33 are separately patentable from the other pending claims for the reasons set forth below.

Claims 23, 25-28, and 30-33 stand rejected under 25 USC 102 in view of Oliver. Applicant respectfully maintains that claim 23 is patentable over Oliver for at least the reason that Oliver fails to disclose, explicitly or inherently, the feature of "providing a comparison means for comparing the value of the memory location of a selected smart pointer giving up its association with the memory-resident element to the value of the next-pointer of the selected smart pointer, to provide a determination whether the linked list contains only the selected smart pointer." (Emphasis Added.) That is, Applicant's claimed test compares two memory values, that of the "memory location of a selected smart pointer giving up its association" to that of the "the next-pointer of the selected smart pointer." Applicant's claimed test does not compare the value of the "next pointer" to that of the "previous pointer", as is done in Oliver. The Examiner concedes that Oliver compares the values of the next and previous pointers when he states that "Oliver discloses the pointers next and previous are examined (compared) before they are deleted from the memory (see col. 5, lines 30-46 and Fig. 5D step 502 and related text)" (Final Action, page 4.) For at least this reason, it is clear that Oliver fails to disclose each and every element recited in claim 23.

Turning more specifically to the portion of the Final Action applying Oliver to claim 23, the Examiner states that

> providing a comparison means for comparing the value of the memory location of a selected smart pointer giving up its association with the memory-resident element to the value of the next-pointer of the selected smart pointer (col. 5, lines 17-20 "the "next pointer"... linked to each other"), to provide a determination whether the linked list contains only the selected smart pointer (col. 5, lines 34-37 "pointer ... examined.. . pointer is the same ... one pointer remaining in the list"). (Final Action at 13.)

But, as the Examiner already acknowledged, the cited text of Oliver at col. 5, lines 34-37 compares the next and previous pointers. The "value of the memory location of a selected smart pointer giving up its association" is not even considered or mentioned in the cited text of Oliver. It is not used in the Oliver test. Only the next and previous pointers are compared in Oliver. Hence, Oliver unquestionably fails to disclose at least Applicant's claimed feature of "comparing the value of the memory location of a selected smart pointer giving up its association with the memory-resident element to the value of the next-pointer of the selected smart pointer." Likewise, Oliver fails disclose the step of "providing a deletion means for deleting the memory-resident element associated with the smart pointer if the value of the next-pointer of the selected smart pointer is equal to the value of the memory location of the selected smart pointer in which the next-pointer is included and not deleting the memory-resident element if the value of the next- pointer of the selected smart pointer is not equal to the value of the memory location of the selected smart pointer in which the next pointer is included" as recited in claim 23. For at least these reason, the rejection of claim 23 in view of Oliver is deficient, as Oliver fails to disclose each and every element recited in claim 23.

Indeed, Applicant has specifically explained to the public in the body of the application reasons not to compare the values of the next and previous pointers in a test for deciding when to delete the object to which a pointer points. To do so would run the risk of creating a dangling pointer, a problem sought to be solved by Applicant's invention. (Application at [004].) Hence, there are significant practical and functional differences between the test claimed and that of Oliver.

Applicant has taught the public that the values of the next and previous pointers are equal when there are two smart pointers on a ring. (Application at [0021].) In addition, Applicant has warned against deleting an object when there are two or more members on a ring, but deleting only when there is one member on the ring, otherwise a dangling pointer will remain on the ring. (Application at [0021] and [0004]). Accordingly, Applicant has taught the public not to use a test comparing the next and previous pointers, but rather a test using either the next or previous pointer and comparing it to the address of the smart pointer to which the next or previous pointer belongs:

> More importantly, a significant distinction exists between the "Next" and "Previous" pointer values of each smart pointer in the single-member ring 100 of Fig. 3 and the two-member ring 100 of Fig. 1. In the two-member ring 100, the "Next" pointer 16 and the "Previous" pointer of the each smart pointer contain the memory address of the other smart pointer. In contrast, for the single-member ring 100, the "Next" pointer 16 and the "Previous" pointer 14 of the first smart pointer 10 each contains the memory address of the smart pointer to which it belongs. Thus, while the values of the "Next" pointer 16 and the "Previous" pointer 14 are equal to each other in both the single-member ring and the two-member ring, the values of the "Next" pointer 16 and the "Previous" pointer 14 are equal to the address of the smart pointer to which they belong only when the ring 100 contains a single-member as shown in Fig. 3. *Therefore, the test for a single-member ring, step 410 of Fig. 4, comprises comparing the value of* either *the "Next" pointer 16 or the "Previous" pointer 14 to the address of the smart pointer to which the "Next" pointer 16 and "Previous" pointer 14 belong, e.g., the first smart pointer 10.* If the test for equality between the value of the "Next" pointer 16 and the address of the first smart pointer 10, as provided at step 410, is true, then the object 30 is deleted at step 420. (Application at [0021]. Emphasis Added.)

In sum, if one were to erroneously conclude that the Oliver test was equivalent to that claimed by Applicant, the "test" would return a 'true' result if there were two smart pointers on the linked list. Then, at the last step of claim 23 the memory-resident element would be prematurely deleted when two pointers remained on the linked list, thereby creating a dangling pointer, in contradiction to what Applicant explicitly teaches and what claim 23 expressly calls for.

Hence, for at least the above reasons, Applicant respectfully submits that Oliver fails to disclose each and every element recited in claim 23, either explicitly or inherently. Accordingly, Applicant respectfully requests that the Board reverse the Examiner's rejection of claim 23, as well as 25-28 and 30-33 which depend respectively therefrom.

7.1.4. <u>Respecting Claims 34</u>

Applicant submits that dependent claim 34 is separately patentable from the other pending claims for the reasons set forth below.

In the Final Action, the Examiner states that "Oliver disclose - wherein the first smart pointer is associated with a first object of a first class and the second smart pointer is associated with a second object of a second class, and wherein the method comprises the step of providing a conversion means for providing automatic conversion between the first smart pointer and the second smart pointer (col. 5, lines 12-22 'second entry ... linked to each other')." (Final Action, pages 14-15.)

Applicant respectfully submits that the text of Oliver cited in the Final Action does not disclose Applicant's claimed feature of "providing a function for automatically converting a smart pointer to an object of a first class to a smart pointer to an object of a second class..." as recited in claim 34. The text of Oliver at column 5, lines 12-22, and as illustrated in Fig. 5B bears no relation to the language of claim 34. Rather, the cited text of Oliver illustrates what happens when the "object is copied" (Column 5, line 12), which is to create "a second entry in the pointer list... <u>for the same object</u>." (Column 5, lines 13. Emphasis Added.) Oliver states that the "second pointer list entry includes a second standard pointer to the <u>*original object*</u>." (Column 5, lines 14-15. Emphasis Added.) That is, Oliver explicitly indicates that both pointer list entries are pointers that point to the <u>*same original object*</u> without regard to the class(es) to which the object might belong. There is only one object disclosed at column 5, lines 12-22 of Oliver, and a single object is not inherently an object of a first-class and an object of a second-class. Therefore, the text at column 5, lines 12-22 fails to disclose, either explicitly or inherently, "an object of a first-class" and "an object of a second-class" as recited in claim 34. Likewise, it follows that Oliver does not disclose Applicant's claimed features of "a smart pointer to an object of a first class" and "a smart pointer to an object of a second class" as recited in claim 34. Still further, Oliver makes no reference at column 5, lines 12-22 to providing a "conversion means for providing automatic conversion..." as recited in claim 34, the absence of which is to be expected, as there is nothing to convert.

Consequently, for all the above reasons, Oliver fails to disclose"wherein the first smart pointer is associated with a first object of a first class and the second smart pointer is associated with a second object of a second class, and wherein the method comprises the step of providing a conversion means for providing automatic conversion between the first smart pointer and the second smart pointer" as recited in claim 34. Since, Oliver fails to disclose each and every element recited in claim 34, Applicant is entitled to patent protection on the subject matter of claim 34. Thus, Applicant maintains that the Examiner has failed to make a showing of anticipation, and Applicant requests that the Board reverse the Examiner's rejection of claim 34 and allow claim 34 to Applicant.

7.1.5. Respecting Claims 35

Applicant submits that claim 35 is separately patentable from the other pending claims for the reasons set forth below.

Claim 35 stands rejected under 25 USC 102 in view of Oliver. Applicant respectfully maintains that claim 35 is patentable over Oliver for at least the reason that Oliver fails to disclose, explicitly or inherently, the feature of "providing a comparison means for comparing the value of the previous pointer to the value of the memory location of the smart pointer in which the selected previous pointer is included, whereby a determination can be made if the ring contains more than one smart pointer." (Emphasis Added.) That is, Applicant's claimed test compares two memory values, that of the "memory location of the smart pointer in which the selected previous pointer is included" to that of the "the previous pointer." Applicant's claimed test does not compare the value of the "next pointer" to that of the "previous pointer", as is done in Oliver. The Examiner concedes that Oliver compares the values of the next and previous pointers when he states that "Oliver discloses the pointers next and previous are examined (compared) before they are deleted from the memory (see col. 5, lines 30-46 and Fig. 5D step 502 and related text)" (Final Action, page 4.) For at least this reason, it is clear that Oliver fails to disclose each and every element recited in claim 35.

Turning more specifically to the portion of the Final Action applying Oliver to claim 35, the Examiner states that

> providing a comparison means for comparing the value of the memory location of a selected smart pointer giving up its association with the memory-resident element to the value of the next-pointer of the selected smart pointer (col. 5, lines 17-20 "the "next pointer"... linked to each other"), to provide a determination whether the linked list contains only the selected smart pointer (col. 5, lines 34-37 "pointer ... examined.. . pointer is the same ... one pointer remaining in the list"). (Final Action at 13.)

But, as the Examiner already acknowledged, the cited text of Oliver at col. 5, lines 34-37 compares the next and previous pointers. The "the memory location of the smart pointer in which the selected previous pointer is included" is not even considered or mentioned in the cited text of Oliver. It is not used in the Oliver test. Only the next and previous pointers are compared in Oliver. Hence, Oliver unquestionably fails to disclose at least Applicant's claimed feature of "comparing the value of the previous pointer to the value of the memory location of the smart pointer in which the selected previous pointer is included." Likewise, Oliver fails disclose the step of "deleting the memory-resident element associated with the smart pointer if the value of the previous pointer of the smart pointer is equal to the value of the memory location of the smart pointer in which the previous pointer is included and not deleting the memory-resident element if the value of the previous pointer of the smart pointer is not equal to the value of the memory location of the smart pointer in which the previous pointer is included" as recited in claim 35. For at least these reason, the rejection of claim 35 in view of Oliver is deficient, as Oliver fails to disclose each and every element recited in claim 35.

Indeed, Applicant has specifically explained to the public in the body of the application reasons not to compare the values of the next and previous pointers in a test for deciding when to delete the object to which a pointer points. To do so would run the risk of creating a dangling pointer, a problem sought to be solved by Applicant's invention. (Application at [004].) Hence, there are significant practical and functional differences between the test claimed and that of Oliver.

Applicant has taught the public that the values of the next and previous pointers are equal when there are two smart pointers on a ring. (Application at [0021].) In addition, Applicant has warned against deleting an object when there are two or more members on a ring, but deleting only when there is one member on the ring, otherwise a dangling pointer will remain on the ring. (Application at [0021] and [0004]). Accordingly, Applicant has taught the public not to use a test comparing the next and previous pointers, but rather a test using either the next or previous pointer and comparing it to the address of the smart pointer to which the next or previous pointer belongs:

> More importantly, a significant distinction exists between the "Next" and "Previous" pointer values of each smart pointer in the single-member ring 100 of Fig. 3 and the two-member ring 100 of Fig. 1. In the two-member ring 100, the "Next" pointer 16 and the "Previous" pointer of the each smart pointer contain the memory address of the other smart pointer. In contrast, for the single-member ring 100, the "Next" pointer 16 and the "Previous" pointer 14 of the first smart pointer 10 each contains the memory address of the smart pointer to which it belongs. Thus, while the values of the "Next" pointer 16 and the "Previous" pointer 14 are equal to each other in both the single-member ring and the two-member ring, the values of the "Next" pointer 16 and the "Previous" pointer 14 are equal to the address of the smart pointer to which they belong only when the ring 100 contains a single-member as shown in Fig. 3. *Therefore, the test for a single-member ring, step 410 of Fig. 4, comprises comparing the value of either the "Next" pointer 16 or the "Previous" pointer 14 to the address of the smart pointer to which the "Next" pointer 16 and "Previous" pointer 14 belong, e.g., the first smart pointer 10.* If the test for equality between the value of the "Next" pointer 16 and the address of the first smart pointer 10, as provided at step 410, is true, then the object 30 is deleted at step 420. (Application at [0021]. Emphasis Added.)

In sum, if one were to erroneously conclude that the Oliver test was equivalent to that claimed by Applicant, the "test" would return a 'true' result if there were two smart pointers on the linked list. Then, at the last step of claim 35 the memory-resident element would be prematurely deleted when two pointers remained on the linked list, thereby creating a dangling pointer, in contradiction to what Applicant explicitly teaches and what claim 35 expressly calls for.

Hence, for at least the above reasons, Applicant respectfully submits that Oliver fails to disclose each and every element recited in claim 35, either explicitly or inherently. Accordingly, Applicant respectfully requests that the Board reverse the Examiner's rejection of claim 35.

Conclusion

In summary, Oliver fails to disclose each and every element of the claims it is alleged to anticipate.

For all the above reasons, the Examiner erred in rejecting claims 3-10, 12, 13, 15-23, 25-28, and 30-35. Thus, the rejections as set forth in the Final Action cannot be maintained. There being no proper grounds for rejection, Applicant is entitled to patent protection on the subject matter of the appealed claims. It is therefore respectfully requested that the rejections of record be reversed and that all pending claims be allowed to Applicant.

Respectfully Submitted,

DANN, DORFMAN, HERRELL AND SKILLMAN

BY: /Niels Haun/
NIELS HAUN
PTO Reg. No. 48,488

Telephone: 215-563-4100
Fax: 215-563-4044

APPENDIX: CLAIM LISTING

1-2. (Canceled.)

3. A computer-implemented method of memory management, comprising the steps of:

providing a smart pointer for association with a memory-resident element, the smart pointer including a next pointer;

providing an assignment means for assigning the next pointer to point to the smart pointer thereby creating a linked list comprising the smart pointer;

providing a comparison means for comparing the value of the next pointer to the value of the memory location of the smart pointer in which the next pointer is included, whereby a determination can be made if the linked list contains more than one smart pointer; and

deleting the memory-resident element associated with the smart pointer if the value of the next pointer of the smart pointer is equal to the value of the memory location of the smart pointer in which the next pointer is included and not deleting the memory-resident element if the value of the next pointer of the smart pointer is not equal to the value of the memory location of the smart pointer in which the next pointer is included.

4. The method according to claim 3, wherein the method comprises the step of providing a common base to the smart pointer.

5. The method according to claim 3, wherein the element is an object in an object-oriented programming environment.

6. The method according to claim 5, wherein the smart pointer includes an object pointer for pointing to the object.

7. The method according to claim 3, wherein the linked list comprises a ring.

8. The method according to claim 3, wherein the smart pointer includes a previous pointer.

9. The method according to claim 8, comprising a step of providing an assignment means for assigning the previous pointer to point to the smart pointer, thereby creating a bi-directional, doubly-linked list.

10. The method according to claim 9, wherein the linked list comprises a ring.

11. (Canceled.)

12. The method according to claim 3, wherein the smart pointer includes a first smart pointer, and wherein the method comprises the step of providing an attachment means for attaching a second smart pointer associated with the memory-resident element to the linked list.

13. A computer-implemented method of memory management, comprising the steps of:

providing a linked list comprising a smart pointer associated with a memory-resident element, the smart pointer including a next-pointer for pointing to the smart pointer;

providing a comparison means for comparing the value of the memory location of the smart pointer to the value of the next-pointer of the smart pointer, to provide a determination whether the linked list contains only the smart pointer; and

deleting the memory-resident element associated with the smart pointer if the value of the next-pointer of the smart pointer is equal to the value of the memory location of the smart pointer in which the next-pointer is included and not deleting the memory-resident element if the value of the next-pointer of the smart pointer is not equal to the value of the memory location of the smart pointer in which the next-pointer is included.

14. (Canceled.)

15. The method according to claim 13, wherein the element is an object in an object-oriented programming environment.

16. The method according to claim 15, wherein the smart pointer includes an object pointer for pointing to the object.

17. The method according to claim 13, wherein the linked list comprises a ring.

18. The method according to claim 13, wherein the smart pointer includes a previous pointer.

19. The method according to claim 18, comprising a step of providing an assignment means for assigning the previous pointer to point to the smart pointer, thereby creating a bi-directional, doubly-linked list.

20. The method according to claim 19, wherein the linked list comprises a ring.

21. The method according to claim 13, comprising a step of providing a deletion means for deleting the memory-resident element associated with the smart pointer if the value of the next-pointer of the smart pointer is equal to the value of the memory location of the smart pointer in which the next-pointer is included.

22. The method according to claim 13, wherein the smart pointer includes a first smart pointer, and wherein the method comprises the step of providing an attachment means for attaching a second smart pointer associated with the memory-resident element to the linked list.

23. A computer-implemented method of memory management, comprising the steps of:

providing a linked list comprising a first smart pointer and a second smart pointer each associated with a memory-resident element, the first smart pointer including a first next-pointer for pointing to the second smart pointer and the second smart pointer including a second next-pointer for pointing to the first smart pointer;

providing a comparison means for comparing the value of the memory location of a selected smart pointer giving up its association with the memory-resident element to the value of the next-pointer of the selected smart pointer, to provide a determination whether the linked list contains only the selected smart pointer; and

providing a deletion means for deleting the memory-resident element associated with the smart pointer if the value of the next-pointer of the selected smart pointer is equal to the value of the memory location of the selected smart pointer in which the next-pointer is included and not deleting the memory-resident element if the value of the next- pointer of the selected smart pointer is not equal to the value of the memory location of the selected smart pointer in which the next pointer is included.

24. (Canceled.)

25. The method according to claim 23, wherein the linked list comprises a ring.

26. The method according to claim 23, wherein the first smart pointer and the second smart pointer each include a previous pointer.

27. The method according to claim 26, comprising a step of providing an assignment means for assigning the previous pointer of the first smart pointer to point to the second smart pointer and for assigning the previous pointer of the second smart pointer to point to the first smart pointer, thereby creating a bi-directional, doubly-linked list.

28. The method according to claim 27, wherein the linked list comprises a ring.

29. (Canceled.)

30. The method according to claim 23, comprising the step of providing an attachment means for attaching a third smart pointer associated with the memory-resident element to the linked list.

31. The method according to claim 23, comprising the step of providing a common base to the smart pointers.

32. The method according to claim 23, wherein the element is an object in an object-oriented programming environment.

33. The method according to claim 32, wherein the first smart pointer and the second smart pointer each include an object pointer for pointing to the object.

34. The method according to claim 32, wherein the first smart pointer is associated with a first object of a first class and the second smart pointer is associated with a second object of a second class, and wherein the method comprises the step of providing a conversion means for providing automatic conversion between the first smart pointer and the second smart pointer.

35. A computer-implemented method of memory management, comprising the steps of:

providing a smart pointer for association with a memory-resident element, the smart pointer including a previous pointer;

providing an assignment means for assigning the previous pointer to point to the smart pointer thereby creating a linked list comprising the smart pointer;

providing a comparison means for comparing the value of the previous pointer to the value of the memory location of the smart pointer in which the selected previous pointer is included, whereby a determination can be made if the ring contains more than one smart pointer; and

deleting the memory-resident element associated with the smart pointer if the value of the previous pointer of the smart pointer is equal to the value of the memory location of the smart pointer in which the previous pointer is included and not deleting the memory-resident element if the value of the previous pointer of the smart pointer is not equal to the value of the memory location of the smart pointer in which the previous pointer is included.

APPENDIX: EVIDENCE

None.

APPENDIX: RELATED PROCEEDINGS

None.